Information-Flow Control by means of Security Wrappers for Active Object Languages with Futures

Farzane Karami[1], Olaf Owe[1], and Gerardo Schneider[2]

[1] Dept. of Informatics, University of Oslo, Oslo {farzanka,olaf}@ifi.uio.no
[2] Dept. of Computer Science and Eng., Chalmers University of Technology, Gothenburg gerardo@cse.gu.se

Abstract. This paper introduces a run-time mechanism for preventing leakage of secure information in distributed systems. We consider a general concurrency language model where concurrent objects interact by asynchronous method calls and futures. The aim is to prevent leakage of secure information to low-level viewers. The approach is based on a notion of *security wrappers*, where a wrapper encloses an object or a component and controls its interactions with the environment. Our run-time system automatically adds a wrapper to an insecure component. The wrappers are invisible such that a wrapped component and its environment are not aware of it.
The security policies of a wrapper are formalized based on a notion of security levels. At run-time, future components will be wrapped upon need, and objects of unsafe classes will be wrapped, using static checking to limit the number of unsafe classes and thereby reducing run-time overhead. We define an operational semantics and sketch a proof of non-interference. A service provider may use wrappers to protect its services in an insecure environment, and vice-versa: a system platform may use wrappers to protect itself from insecure service providers.

Keywords: Active objects · Futures · Information-flow security · Non-interference · language-based security · Distributed systems

1 Introduction

Given the large number of users and service providers involved in a distributed system, security is a critical concern. It is essential to analyze and control how confidential information propagates between nodes. When a program executes, it might leak secure information to public outputs or send it to malicious nodes. *Information-flow control* approaches track how information propagates during execution and prevent leakage of secure information [24]. Program variables are tagged typically with security levels; such as *high* and *low*, to indicate secure and public data. In this setting, an "attacker" could be seen as a low-level object that is not supposed to see high information. The basic semantic notion of information-flow security is *non-interference* [10]. This means that in any two

executions of a program, if high inputs are changed, but low inputs are the same, then the low outputs will be the same (at least for locally deterministic programs). This way, an attacker (a low object) cannot distinguish between observable behaviors of the two executions since low outputs are independent of the high inputs [12].

We will consider a high-level model for object-oriented distributed systems suited for service-oriented systems, namely the *active object model* [5]. Method interaction is implemented by message passing; moreover, most active object languages support a communication paradigm called *futures* [5]. A future is a component that is created by a remote method call and eventually will contain the corresponding return value [3]. Therefore, the caller does not need to block while waiting to get the return value: it can continue with other tasks and later get the value from the corresponding future. Futures can be passed to other objects, called *first-class futures*. In this case, any object that has a reference to a future can access its content, which may be a security threat if the future contains secure data. Futures offer a flexible way of communication and sharing results, but handling them appropriately in order to avoid security leakages requires run-time checking (described in Sec. 2.1).

Our goal is to design a permissive and precise security mechanism for controlling object communications in active object languages supporting first-class futures. Our security mechanism is inspired by the notion of *wrappers* in [21], where a wrapper encloses an object and enforces safety rules. In the present paper, we suggest a notion of *security wrapper*, which wraps an object or a future at run-time and performs security controls. Such wrappers are added by the operational semantics upon need, and a wrapped component and its environment are not aware of the presence of the wrapper. Security wrappers block object communications that lead to leakage of secure data to low objects. A future is wrapped if it contains a high value, and the wrapper blocks illegal access by low objects. The operational semantics of a wrapper is defined based on run-time security levels, resulting from a flow-sensitive information-flow enforcement [23]. We enrich the operational semantics with dynamic information-flow rules [23] where security levels of variables are allowed to change after an assignment. Therefore, our dynamic approach guarantees a degree of permissiveness and is precise since it deals with the exact run-time security levels.

The operational semantics of our security framework is provided in the style of Structured Operational Semantics (SOS). In order to minimize run-time overhead, we suggest static analysis to limit the number of classes where security checking and wrappers are needed since often only a few methods deal with secure information. In the resulting hybrid approach, the static analysis determines which classes cannot produce any high output, so-called *safe classes*, while the run-time system takes care of the precise security checking of objects of unsafe classes and futures created by such objects. Assuming a sound static analysis, we show that our proposed hybrid approach ensures the non-interference property.

In summary, our contributions are: i) a notion of security wrappers for enforcing noninterference and security control in object interactions (Sec 4), ii)

the use of static analysis to reduce the run-time overhead (Sec 4.1), iii) defining the operational semantics for the dynamic information-flow enforcement with automatic deployment of wrappers (Secs. 4.2, 4.3) for our language (Sec 4), and iv) an outline of the proof that our approach satisfies non-interference.

2 Background

Information-flow control approaches detect illegal flows. During program execution, there are two kinds of leakage of information, namely *explicit* and *implicit flows* [24]. For simplicity, we assume two security levels, L (low) and H (high). In the setting with observable and non-observable variables, an explicit flow happens when assigning a low variable (l) with a high value (h) by $l := h$. In the setting without observable variables, one may deal with this by letting the level of l be dynamically changed to H. In an implicit flow, there is an indirect flow due to control structures. For example, in the `if` statement: $l := 0$ `if` h `then` $l := 1$ `fi`, the guard h is high, and it affects the value of l indirectly. In order to avoid implicit flows, a program-counter label (pc) is introduced [24]. If the guard is high, then pc becomes high, indicating a high context. (In run-time analysis, one may use a stack to deal with nested control structures.)

Information-flow control approaches are divided into two categories, static and dynamic [23]. Static analysis is conservative [12]: to be sound, it over-approximates security levels of variables (for example, it over-approximates a formal parameter to high, while at run-time, a corresponding actual parameter can be low). This causes unnecessary rejections of programs, especially when the complete program is not statically known, as is usually the case in distributed systems. On the other hand, static analysis has less run-time overhead since security checks are performed before program execution [12]. *Dynamic information-flow* techniques perform security checks at run-time, and this introduces overhead. But they are more permissive and precise since they deal with the exact security levels instead of an over-approximation [12].

For example, consider the following method body:

$\{$`if` low_test `then` $x := high\_exp$ `else` $x := low\_exp$ `fi`; `return` $x\}$

where low_test and low_exp evaluate to low values, while $high\_exp$ evaluates to a high value. A sound static analysis will detect a high method result here since the value of low_test is not known; while at run-time, an execution of the method may give a low result (when low_test evaluates to *false*). The example shows that static analysis over-approximates the security level, in contrast to run-time analysis. Similarly, the parameter mechanism gives rise to static over-approximation. For a method T $triv(T\ x)\{$`return` $x\}$, where T is a type containing both high and low values, static analysis will detect a (potentially) high result, whereas for calls with a low input value, the result is detected as low at run-time. However, this could be handled by multiple static method profiles as in [19] (when low T values are reflected by a subtype of T). For first-class futures, the situation is worse: a get statement on a future is detached from the call statement and also from the method name. Therefore, the static analysis of a get statement must over-approximate the level of the possible future values,

while the exact level is revealed during run-time. This means that static analysis of security levels in languages with first-class futures can easily lead to a high degree of over-approximation.

In what follows, we briefly explain some of the terminologies of information-flow security that we use in this paper:

Security levels. Variables are tagged with security levels, organized by a partial order $\sqsubseteq$ and a join $\sqcup$ operator, such that $L \sqsubseteq H$ and $L \sqcup H = H$. The $\sqcup$ operator returns the least upper bound of two security levels. Inside a class, declarations of fields, class parameters, and formal parameters may have statically declared initial security levels. These levels may change with statements. We define a new syntax for object creation to assign security levels to objects.

Flow-sensitivity. By a *dynamic flow-sensitive analysis*, security levels of variables propagate to other variables, and precise levels are evaluated during execution. Variables start with their declared security levels (the ones without levels are assumed as L), but levels may change after each statement. In an assignment, the left-hand-side level becomes high if *pc* is high, or there is a high variable on the right-hand-side. The left-hand-side level becomes low if *pc* is low, and there is no high variable on the right-hand-side [23]. Otherwise, the security level of a variable does not change. E.g., a flow-sensitive analysis accepts the program $h := 0$; `if` h `then` $l := 1$ `fi`; `return` l; since the level of h is updated to L after the first assignment, hence there is no leakage. In `if` statements, in order to avoid implicit flows, when the guard is high, the security levels of variables appearing on the left-hand side of assignments in the taken and untaken branches are raised to high [23]. E.g., considering an initial environment $\Gamma = \{h \mapsto H, l_1 \mapsto L, l_2 \mapsto L\}$ and the program: `if` h `then` $l_1 := 1$ `else` $l_2 := 0$ `fi` when the condition is true, Γ changes to $\Gamma = \{h \mapsto H, l_1 \mapsto H, l_2 \mapsto H\}$ for a sound flow-sensitive analysis [23]. In a dynamic approach, in order to have a sound flow-sensitive analysis, the assigned variables in the untaken branches should be given to the analysis, which can be provided by static analysis of the program code [23,4].

2.1 Active object languages

Active object languages are based on a combination of the *actor model* [1] and object-oriented features [5]. Some well-known active object languages are Rebeca [25,26], Scala/Akka [11,27], Creol [14], ABS [13], Encore [6], and ASP/ProActive [8,7]. In communication with futures, when a remote method call is made, a future object with a unique identity is created. Futures can be explicit with a specific type and access operations like in ABS or can be implicit with automatic creation and access [5]. E.g., in ABS, explicit futures are created as in $Fut[T]\ f := o!m(\bar{e}); v := f.\texttt{get}$, where f is a future variable of type $Fut[T]$, and T is the type of the future value. The symbol "!" indicates an asynchronous method call m of object o with actual parameters $\bar{e}$, and the future value is retrieved with a `get` construct when needed. The variable f can be passed to other objects as a parameter (first-class futures). The caller may continue with other processes while the callee is computing the return value. The callee sends back the return value to the corresponding future, and then the future is called

Basic constructs	
$x := \texttt{new}_{lev}\ c(\overline{e})$	object creation with the security level *lev*
`return` e	creating a method result/future value
`if` b `th` s [`el` s'] `fi`	if statement (b a Boolean condition)
$f := o!m(\overline{e})$	remote asynchronous call, future variable f
$x := f.\texttt{get}$	blocking access operation on future f
$o!m(\overline{e})$	simple asynchronous remote call

Fig. 1. Statement syntax. Here $\overline{e}$ is an expression list. Brackets denote optional parts.

resolved. A synchronous call is denoted by $o.m(\overline{e})$, which blocks the caller until the return value is retrieved.

Information-flow security with futures. Static analysis is in general difficult for programs with futures, where the result of a call is no longer syntactically connected to the call, compared to the call/return paradigm in languages without futures [15]. For example, a future may be created in one module and received as a parameter in another. Thus, a future may not statically correspond to a unique call statement. One could overestimate all future values as high, but this would severely restrict the set of acceptable programs. It would be better to overestimate the set of possible call statements that corresponds to a given get statement, but this requires access to the whole program, which is often problematic for distributed systems. Moreover, the return values of these overestimated calls may have different security levels, which also results in overestimation.

A static analysis that assumes references as low, allows passing of future references. However, the exact security level of a future value is revealed when it becomes resolved, which goes beyond static analysis. For example, if a low-level object performs $x := f.\texttt{get}$, and f refers to a future with a high value, it is a leakage of information. A dynamic approach is required to control access to a future value at run-time when it is resolved, and if the value is high it needs protection. The futures concept makes static checking less precise, and the need for complementary run-time checking is greater, as provided in the present paper.

3 Our core language

In order to exemplify our security approach, the security semantics (in Sec. 4.2) is embedded in a simple, high-level core language. All remote calls are made by means of futures, where the method result is always returned to the corresponding future. Figure 1 gives the syntax of statements. The statement $f := o!m(\overline{e})$ is an asynchronous call with futures, and $o!m(\overline{e})$ is an asynchronous call without waiting for the result and associating a future. We define an extended syntax for object creation $\texttt{new}_{lev}\ c(\overline{e})$, where *lev* is the object's level (it can be L or H).

Figure 2 illustrates a health care service in our core language, involving futures for the sharing of secure medical records. Personnel and patients with lower-level access are not allowed to access medical records. High variables are

```
1  data type Result = ... // definition of medical data
2  interface ServiceI { Void produce() ... }
3  interface ProxyI { Void publish(Fut[Result] q, PatientI a, List[PersonnelI] d) ... }
4  interface LabI { Result_H search(PatientI a) ... }
5  interface PatientI { Void send(Result_H r) ... }
6  interface PersonnelI { Void send(Result_H r) ... }
7  interface DataBase { ... }
8  class Service(LabI lab, DataBase db) implements ServiceI {
9   ProxyI proxy = new_H Proxy(this);
10  this!produce();   // initial action, starting a produce cycle
11  Void produce() { Fut[Result] f;  PatientI a;   List[PersonnelI] d = Nil;
12    ....     // finding a patient and the associated personnel in a database
13    f:=lab!search(a); // searching for the test result of patient a
14    proxy!publish(f, a, d); } } //sending the future f, not waiting for the result
15
16 class Proxy(ServiceI s) implements ProxyI{ Result_H x;
17  Void publish(Fut[Result] f, PatientI a, List[PersonnelI] d) {
18  x:=f.get;   // waiting for future and assigning the value to x. x becomes H
19  a!send(x);    // x is now H
20  d!send(x);   // multicasting, x is H
21  s!produce(); } }
```

Fig. 2. Example of sharing *high* patients' test results by means of futures

emphasized based on user specifications, in this case reflecting patients' medical test results. The server, specified by the class *Service*, searches for a patient's test result, and the object *proxy* publishes the result to the patient and personnel. In Fig 2, in line 10, a produce cycle is initiated between the *server* and *proxy*. In line 13, the server searches for the test result of a patient with the userId a by sending a remote asynchronous call to the laboratory $f := lab!search(a)$, where f is the future variable. In line 14, the server calls $proxy!publish(f, a, d)$ and passes the future f, userId a, and personnelId d to the object *proxy*. Both *search* and *publish* are asynchronous calls, thus the server does not wait for the return values and is free to respond to any client request. In line 18, the object *proxy* waits for the test result and assigns the result to variable x by performing $x := f.\texttt{get}$. Then *proxy* sends x to the patient and personnel.

A static analysis over-approximates the security levels of test results as high, which leads to rejections of information passing. Note that the two *send* calls in the class *Proxy* would not be allowed if we only use static checking since we cannot tell which patients and personnel have a high enough level. A static analysis which considers references as low allows passing the future f to the object *proxy* (line 14), but later when it is resolved, the future value can be high, and the *proxy* compromises security by sending this value to other objects.

4 A framework for non-interference

Like Creol, our core language is equipped with interface encapsulation, which means that created objects are typed by interfaces, not classes [14]. As a result, remote access to fields or methods that are not declared in an interface is impossible. Therefore, observable behavior of an object is limited to its interactions through remote method calls. Illegal object interactions are the ones leading to an information-flow from high information to low level objects. An object can reveal confidential information in method calls by sending actual parameters with high security levels to low-level objects. If a future contains data with a high security level, low-level objects' access is illegal.

We exploit the notion of wrappers to perform dynamic checking for enforcing non-interference in object interactions. A wrapper blocks illegal interactions. Wrappers' security policies are based on run-time security levels. Inside an object, in order to compute the exact security levels of created messages or return values, the flow-sensitivity must be active. The operational semantics for the dynamic flow-sensitive analysis, is given in Sec. 4.2 and for wrappers, in Sec. 4.3.

We can be conservative and wrap all objects and correspondingly activate flow-sensitivity, but this will cost run-time overhead. In order to be more efficient at run-time, it is important to perform dynamic checking only for components where it is necessary. We benefit from static analysis to categorize a class definition as *safe* or *unsafe*. *A class is safe if it does not have any method calls with high actual parameters and return values. A class is unsafe if it has a method call with at least one high actual parameter or a high return value.* Objects created from unsafe classes are wrapped, and flow-sensitivity will be active inside these objects. Objects from safe classes do not need a wrapper or active flow-sensitivity. This will make the execution of objects of safe classes faster, as we avoid a potentially large number of run-time checks and wrappers.

4.1 Static analysis

Our security approach can be combined with a sound static over-approximation for detecting security errors and safe classes, e.g., the one proposed in [20], which is more permissive (to classify a class as safe) than the static analysis indicated here, in that high communication is considered secure as long as the declared levels of parameters are respected. In a class, variables are declared with maximum security levels (the maximum level that can be assigned at run-time). The same for future variables at the time of declaration, for example, $Fut[T_H]\ x$ indicates that x is a high future variable. Local variables without a declared security level start with the level L (as default) but may change after each statement due to the flow-sensitivity. Dataflow typing rules inside an object can be defined similar to [20]; however, we change the typing rules for method calls and return values to classify unsafe and safe classes. A class is defined as safe if the confidentiality of each method is satisfied. The confidentiality of a method is satisfied if the typing rules for its return value and actual parameters are satisfied. The typing rules check that each occurrence of an actual parameter and a return value are not high; then, the class is safe; otherwise, it is unsafe

config	::= ϵ | object | flowsen-obj | msg | future | wrapper | class | config config	
object	::= $ob(o, a, p, lev)$	$d ::= v \mid v_{lev}$
flowsen-obj	::= $ob(o, a, p, lev, pcs)$	$p ::= (l, s) \mid idle$
msg	::= $invc(f, m, \bar{d}, o)_{lev} \mid comp(d, f)_{lev}$	
future	::= $fut(f, d)$	
wrapper	::= $Wr\{wId, lev \mid$ config $\}$	
class	::= $Cl(c \mid a', mm)_{lev}$	

Fig. 3. The components of a configuration.

and needs dynamic checking. The typing rule for getting a future, checks that if a future variable is high, then the class is classified as unsafe. Alternatively, we could have used another sound static analysis, for instance (the relevant parts of) the static analysis defined for ABS in [22], and adapt it to our setting.

We categorize safe and unsafe classes for the example in Fig. 2. The interface laboratory *LabI* has a method with a high return value (*search*). Thus the object *lab* is unsafe and flow-sensitivity is active to compute the security level of the return value at run-time. The class *Proxy* is unsafe since it has at least one method call with a high actual parameter ($a!send(x)$), thus object *proxy* is active flow-sensitive and wrapped.

4.2 Security semantics

We here discuss the operational semantics of our core language with the embedded notions of flow-sensitivity and security wrappers in Figs. 4, 5. The small-step operational semantics is defined by a set of rewrite rules [17]. In a rule, premises are above the line and one step rewrite is under the line. A rule is applied to a subset of a configuration if the premises are satisfied, and the subset is changed from the left-hand-side to the right-hand-side of the rewrite rule.

In Fig. 3, an execution state is modeled as a configuration **config**, which is a multiset of objects (with or without active flow-sensitivity), messages, futures, wrappers, and classes. (Classes are included in a configuration to provide static information about fields and methods.) An **object** is represented as: $ob(o, a, p, lev)$, where o is the object identity, a is the field state, p is the current *active process*, and lev is the object's level ($lev \in \{L, H\}$). An active process p is a pair (l, s), where l is the local variables state, and s is a list of statements, or it is $idle$ representing an empty local state and no statements. A state is a mapping (substitution) binding variables to values. A **flowsen-obj** represents an flow-sensitive object with an extra field pcs that denotes a stack of context security levels inside an object, where $pcs = emp$ denotes an empty stack.

A **class** is represented as: $Cl(c \mid a', mm)_{lev}$, where c is the class name, a' is the initial state of the class fields (attributes), mm is a multiset of method declarations (with local variables and code), and lev denotes the type of the class, i.e., if $lev = L$, the class is safe, and if $lev = H$, the class is unsafe. A **msg** represents an invocation message or a completion message. In an invocation

message, f is the future identity, m is the method name, $\bar{d}$ is a list of actual parameters, and *lev* is a level attached to the message at time of creation. If a message is created in a high context, then $lev = H$; otherwise, $lev = L$. A completion message contains a return value d and a future identity f, and *lev* represents the context level. The notation d denotes a value v or a value with security level v_{lev}. The **future** component shows a resolved future with identity f and the value d, and $fut(f, \_)$ denotes an unresolved future. A **security wrapper** is represented as: $Wr\{wId, lev \mid config\}$, where *wId* is the wrapper's identity, *lev* is the level, and *config* denotes the configuration inside the wrapper.

Auxiliary functions. Let Γ be a mapping and $[x \mapsto d]$ be a binding, mapping x to d. The notation $\Gamma[x \mapsto d]$ represents the update of Γ with the binding. The look-up function is represented as $\Gamma(x)$, where $\Gamma[x \mapsto d](x) = d$. The map composition $a\#l$ indicates that the binding of a variable in the inner scope l shadows any binding of that variable in the outer scope a. Thus $a\#l(x)$ gives $l(x)$ when defined, otherwise $a(x)$. Consider an object with attribute state a and local state l. Then the composition $a\#l$ defines the *object state*. The notation $[\![e]\!]$ denotes the evaluation of expression e, where variables are evaluated according to the object state. The evaluation in $[\![e]\!]$ is strict in the sense that the resulting level is high if e contains variables that have a high security level. Other auxiliary functions are given as follows:

- The function $level(d)$ returns the security level of d, such that $level(v_{lev}) = lev$, and for an untagged value $level(v) = L$. If $\overline{e}$ is a list of expressions, then $[\![\overline{e}]\!] = \bar{d}$ returns a list of data, and $level(\bar{d}) = \sqcup\ level(d_i), \forall d_i \in \bar{d}$ (the join of all data in $\bar{d}$).
- The function $level(o)$ returns the level of the object o.
- The function $level(pcs)$ returns the join of security levels in *pcs*, where if $pcs = emp$, $level(pcs) = L$, and if $pcs \neq emp$, $level(pcs) = H$.
- The function $update_H(s)$ raises the security levels of variables appearing in the left-hand-side of assignments in s to high.
- The function *fresh*() returns a unique identity for an object or a future.
- The function $bind(o, m, \bar{d}, f)$ returns a process, where the method m in the class of the object o is activated, and the method' parameters are bound to the actual ones ($\bar{d}$), and a reserved local variable *label* is bound to f, denoting where to send the return value of the method [13].
- The function $bind(o, m, \bar{d})$ returns a process without the binding for the *label*, in case the method's result is not needed.
- The function $safe(Cl\,(c \mid a, mm)_{lev})$ returns true if $lev = L$ and false otherwise.

Figure 4 represents the flow-sensitivity semantics of objects. The NEW rule shows the command $x := \texttt{new}_{lev} c'(\overline{e})$ in the active process of an object o, where c' is an unsafe class. The rule creates an active flow-sensitive object o' and a wrapper and assigns o' to x. The active process of the new object o' is initially *idle*, denoting an empty active process. The level of o' is *lev* as it is specified in the command $\texttt{new}_{lev} c'(\overline{e})$, if not, the level is assumed low. The stack of *pcs* is empty, denoted by *emp*. The wrapper has the same identity (o') and the level

$$\textsc{new}\quad \frac{o' = \mathit{fresh}()\quad \mathit{false} = \mathit{safe}(C')}{ob(o,a,(l,x := \texttt{new}_{lev}\ c'(\bar{e});s),lev') \rightarrow ob(o,a,(l,x := o';x!\mathit{init}(\bar{e});s),lev')\ \ Wr\{o',lev \mid ob(o',a'[\mathit{this} \mapsto o'],\mathit{idle},lev,\mathit{emp})\}}$$

$$\textsc{assign-local}\quad \frac{x \in \mathit{dom}(l)\quad v_{lev'} = [\![e]\!]}{ob(o,a,(l,x := e;s),lev,pcs) \rightarrow ob(o,a,(l[x \mapsto v_{lev' \sqcup level(pcs)}],s),lev,pcs)}$$

$$\textsc{assign-attribute}\quad \frac{x \in \mathit{dom}(a)\quad v_{lev'} = [\![e]\!]}{ob(o,a,(l,x := e;s),lev,pcs) \rightarrow ob(o,a[x \mapsto v_{lev' \sqcup level(pcs)}],(l,s),lev,pcs)}$$

$$\textsc{if-low-true}\quad \frac{\mathit{true}_L = [\![e]\!]}{ob(o,a,(l,\texttt{if}(e)\ s'\ \texttt{el}\ s''\ \texttt{fi};s),lev,pcs) \rightarrow ob(o,a,(l,s';s),lev,pcs)}$$

$$\textsc{if-low-false}\quad \frac{\mathit{false}_L = [\![e]\!]}{ob(o,a,(l,\texttt{if}(e)\ s'\ \texttt{el}\ s''\ \texttt{fi};s),lev,pcs) \rightarrow ob(o,a,(l,s'';s),lev,pcs)}$$

$$\textsc{if-high-true}\quad \frac{\mathit{true}_H = [\![e]\!]}{ob(o,a,(l,\texttt{if}(e)\ s'\ \texttt{el}\ s''\ \texttt{fi};s),lev,pcs) \rightarrow ob(o,a,(l,s';\mathit{endif}(s'');s),lev,pcs.\mathit{push}(H))}$$

$$\textsc{endif}\quad \frac{l' = l[\mathit{update}_H(s')]}{ob(o,a,(l,\mathit{endif}(s');s),lev,pcs) \rightarrow ob(o,a,(l',s),lev,pcs.\mathit{pop}())}$$

$$\textsc{if-high-false}\quad \frac{\mathit{false}_H = [\![e]\!]}{ob(o,a,(l,\texttt{if}(e)\ s'\ \texttt{el}\ s''\ \texttt{fi};s),lev,pcs) \rightarrow ob(o,a,(l,s'';\mathit{endif}(s');s),lev,pcs.\mathit{push}(H))}$$

$$\textsc{call-fut}\quad \frac{f = \mathit{fresh}()\quad o' = [\![e]\!]\quad \bar{d} = [\![\bar{e}]\!]}{ob(o,a,(l,x := e!m(\bar{e});s),lev,pcs) \rightarrow ob(o,a,(l,x := f;s),pcs)\ \ \mathit{fut}(f,\_)\ \ \mathit{invc}(f,m,\bar{d},o')_{level(pcs)}}$$

$$\textsc{call}\quad \frac{o' = [\![e]\!]\quad \bar{d} = [\![\bar{e}]\!]}{ob(o,a,(l,e!m(\bar{e});s),lev,pcs) \rightarrow ob(o,a,(l,s),lev,pcs)\ \ \mathit{invc}(m,\bar{d},o')_{level(pcs)}}$$

$$\textsc{start-fut}\quad \frac{p = \mathit{bind}(o,m,\bar{d},f)}{ob(o,a,\mathit{idle},lev,pcs)\ \ \mathit{invc}(f,m,\bar{d},o)_{lev'} \rightarrow ob(o,a,p,lev,pcs.\mathit{push}(lev'))}$$

$$\textsc{start}\quad \frac{p = \mathit{bind}(o,m,\bar{d})}{ob(o,a,\mathit{idle},lev,pcs)\ \ \mathit{invc}(m,\bar{d},o)_{lev'} \rightarrow ob(o,a,p,lev,pcs.\mathit{push}(lev'))}$$

$$\textsc{return}\quad \frac{d = [\![e]\!]\quad f = l(\mathit{destiny})}{ob(o,a,(l,\texttt{return}(e);),lev,pcs) \rightarrow ob(o,a,\mathit{idle},lev,pcs)\ \ \mathit{comp}(d,f)_{level(pcs)}}$$

Fig. 4. Flow-sensitive operational semantics, $lev, lev' \in \{l, H\}$.

(lev) of the object o'. The semantics of the actual class parameters is treated like parameters of an asynchronous call $x!\mathit{init}(\bar{e})$ (creating an invocation message by the rule CALL), where *init* is the name of the initialization method of a class. Note that if $\bar{e}$ contains high security level data, the wrapper does not send the corresponding invocation message to the new object if the new object is low-level (see rule WR-INVC-ERROR in Fig. 5, which we explain later). The Rule ASSIGN-LOCAL shows an assignment $x := e$, where x is in the local state l, e is evaluated to $v_{lev'}$, and x is updated in l with the new value v and the level $lev' \sqcup \mathit{level}(pcs)$. Therefore, the level of x is updated with the right-hand-side level joined with that of pcs. In IF-LOW-TRUE, the guard's security level is low, and the guard is true (true_L), thus the corresponding branch s' is taken. While

in IF-LOW-FALSE, since the guard is false, the else branch s'' is taken. In IF-HIGH-TRUE and IF-HIGH-FALSE, since the guard's security level is high, similar to the approach in [23], the security levels of variables appearing in assignments in both branches are raised to high to avoid implicit flows. In the rules, the guard's security level H is pushed to the *pcs* stack, resulting in a high security context, where all the messages created in a high context will have high security levels (see rules CALL-FUT, CALL). Moreover, assignments in the taken branch result in high security levels (see ASSIGN-LOCAL and ASSIGN-ATTRIBUTE). The added statement $endif(s'')$, where s'' is the untaken branch, marks the join point of the `if` structure and raises the assigned variables' levels in the untaken branch. In the ENDIF rule, the function $update_H(s'')$ raises the security levels of variables appearing in the left-hand-side of assignments in s'' to high, and these variables are updated in the local state. Moreover, the last element of *pcs* is removed ($pcs.pop()$), reflecting the previous context level.

In the rules, we do not cover local calls, which do not involve object interactions (therefore, less interesting here). The CALL-FUT rule deals with an asynchronous call $x := e!m(\overline{e})$, where x is a future variable, and e is the callee. The call generates a (not resolved) future with a unique identity f, where f is assigned to x, and an invocation message containing f, m, actual parameters $\bar{d}$, and the callee o'. The invocation message's level is $level(pcs)$, which is needed to avoid indirect leakage from the caller. The rule CALL shows an asynchronous call $e!m(\overline{e})$ without an associated future, where the method's result is not needed. The call creates an invocation message containing m, $\bar{d}$, and the callee o', and the message' level is $level(pcs)$. The START-FUT rule is applied when an object is *idle*, and there is an invocation message to the object. The object's active process is updated with p, which is the *bind*'s result, where method m is activated, formal parameters are bound to the actual ones ($\bar{d}$), and the local variable *label* is bound to the future identity (f) for sending the method's result to the future by a `return` statement. The level of the received message lev' is added to the object's stack *pcs*. This avoids implicit leakage from the sender. In the START rule, the invocation message does not contain a future identity, and the object starts execution the corresponding method, which is activated by the *bind* function without the binding for the *label* variable. The RETURN rule interprets a `return` statement, which creates a completion message to the corresponding future, which is looked up in the local state ($l(label)$), and the object becomes *idle*. The security level of the completion message is $level(pcs)$ to avoid indirect leakage from the callee to the recipients of the future value. We assume that each method body ends with a `return` statement. The rules for objects without active flow-sensitivity are similar but without security levels, *pcs*, and wrappers.

4.3 Operational semantics of security wrappers

In this section, we discuss the operational semantics of security wrappers. As mentioned, a wrapper for an object is created in the rule NEW in Fig.4. A wrapper has the same identity as the wrapped component; thereby, the wrapper represents the component to the environment. Invocation messages generated

$$\textsc{wr-invc}\quad \frac{lev' \sqcup level(\bar{d}) \sqsubseteq level(o')}{Wr\{o, lev \mid invc(f, m, \bar{d}, o')_{lev'}\ config\ \} \rightarrow Wr\{o, lev \mid config\ \}\ invc(f, m, \bar{d}, o')_{lev'}}$$

$$\textsc{high-fut}\quad \frac{H = level(d) \sqcup lev}{fut(f, \_)\ \ comp(d, f)_{lev} \rightarrow Wr\{f, H \mid fut(f, d)\ \}}$$

$$\textsc{wr-invc-error}\quad \frac{lev' \sqcup level(\bar{d}) \sqsupset level(o')}{Wr\{o, lev \mid invc(f, m, \bar{d}, o')_{lev'}\ config\ \}\ fut(f, \_) \rightarrow Wr\{o, lev \mid config\ \}\ fut(f, \texttt{error})}$$

$$\textsc{low-fut}\quad \frac{L = level(d) \sqcup lev}{fut(f, \_)\ \ comp(d, f)_{lev} \rightarrow fut(f, d)}$$

$$\textsc{error-fut}\quad \frac{f = [\![e]\!]}{fut(f, \texttt{error})\ ob(o, a, (l, x := e.\texttt{get}; s), lev, pcs) \rightarrow fut(f, \texttt{error})\ \ ob(o, a, (l, x := \texttt{error}; s), lev, pcs)}$$

$$\textsc{high-fut}\quad \frac{H = level(d) \sqcup lev}{fut(f, \_)\ \ comp(d, f)_{lev} \rightarrow Wr\{f, H \mid fut(f, d)\}}$$

$$\textsc{invc-wr}\quad \frac{\forall lev_i \in level(\bar{d}) : lev_i \sqsubseteq \Lambda[m, i]}{Wr\{o, lev \mid config\ \}\ invc(f,\ m, \bar{d}, o)_{lev'} \rightarrow Wr\{o, lev \mid invc(f, m, \bar{d}, o)_{lev'}\ config\}}$$

$$\textsc{invc-wr-error}\quad \frac{\exists lev_i \in level(\bar{d}) : lev_i \sqsupset \Lambda[m, i]}{Wr\{o, lev \mid config\ \}\ invc(f,\ m, \bar{d}, o)_{lev'} \rightarrow Wr\{o, lev \mid config\}}$$

$$\textsc{error-high-fut-get}\quad \frac{f = [\![e]\!]\ \ lev \sqsubset H}{Wr\{f, H \mid fut(f, d)\}\ ob(o, a, (l, x := e.\texttt{get}; s), lev, pcs) \rightarrow Wr\{f, H \mid fut(f, d)\}\ ob(o, a, (l, x := \texttt{error}; s), lev, pcs)}$$

$$\textsc{high-fut-get}\quad \frac{f = [\![e]\!]\ \ lev \sqsupseteq H}{Wr\{f, H \mid fut(f, d)\}\ ob(o, a, (l, x := e.\texttt{get}; s), lev, pcs) \rightarrow Wr\{f, H \mid fut(f, d)\}\ ob(o, a, (l, x := d; s), lev, pcs)}$$

$$\textsc{low-fut-get}\quad \frac{f = [\![e]\!]}{fut(f, d)\ ob(o, a, (l, x := e.\texttt{get}; s), lev, pcs) \rightarrow fut(f, d)\ ob(o, a, (l, x := d; s), lev, pcs)}$$

Fig. 5. Operational semantics involving wrappers, $lev, lev' \in \{l, H\}$.

by the CALL-FUT and CALL rules will first meet the object's wrapper for security checking before being sent to the callee. The WR-INVC rule in Fig. 5, represents a wrapper with an invocation message inside, which is produced by the object o. If the join ($\sqcup$) of the message's level lev' and the actual parameters' levels $level(\bar{d})$ is less than or equal to the destination object' level ($level(o')$), then the wrapper allows the message to go out. In WR-INVC-ERROR, since the recipient object's level is less than the message's level, the invocation message is deleted and the corresponding future value is replaced by an error value. This can be combined with an exception handling mechanism such that an exception is raised when a get operation tries to access an error value. However, as this is beyond the scope of this paper, we ignore the exception handling part. We simply indicate exceptions by assignments with `error` in the right-hand-side. The ERROR-FUT rule represents the case where a future value is `error`; the object performing the get command $x := e.\texttt{get}$, where e refers to the future, assigns an error to x. The rule ASSIGN-ATTRIBUTE shows an assignment, where x is in the object's fields.

The INVC-WR rule represents a wrapper and an incoming invocation message to the object o. The notation $\Lambda[m, i]$ indicates the level of the ith formal parameter of the method m as declared in the class. If the security level of each actual parameter (lev_i) is less than or equal to the security level of the corresponding formal parameter, then the wrapper allows the message to go through and adds it to its configuration inside. Otherwise, the invocation message is deleted in INVC-WR-ERROR. In LOW-FUT, an unresolved future gets the corresponding completion message containing d, hence the future becomes resolved with d. The join of the message's level lev and $level(d)$ is low, thus no wrapper is created. In HIGH-FUT, $lev \sqcup level(d) = H$, thus the future becomes wrapped and resolved. Since the future is high, a wrapper is created to protect it, and the wrapper has the same identity and level as the future. The ERROR-HIGH-FUT-GET rule represents a wrapped future and an object that wants to get the future value. If the security level of the object (lev) asking for the value is less than the wrapper (H), then the wrapper sends an error value. In HIGH-FUT-GET, the object gets the value from the wrapped future since the object's level is greater than or equal to H. The LOW-FUT-GET rule shows that an object gets the value from an unwrapped future without security checking.

4.4 Non-interference

We show that our security framework satisfies non-interference. Non-interference considers the observable behavior of different executions. The *observable behavior* of an object consists of invocation messages and completion messages. Even the observable behavior of object creation, by the NEW rule in Fig. 4, is an asynchronous call $x!init(\bar{e})$, which creates an invocation message. Since object and future identities may change from execution to execution, we must compare executions relative to a correspondence of such identities in one execution to those in another execution. Corresponding objects must be of the same class.

A message is said to be low if it does not have a high tag nor contain any parameters with high tags. Two low messages are *indistinguishable*, $\simeq$, if the identities in the messages correspond to each other, and other values are equal. Two execution states of corresponding objects o and o' are said to be *indistinguishable* if the values of their local variables and attributes are indistinguishable and they have the same remaining statement lists, and also agree on other system variables, including flow sensitivity (with same values of pcs).

Definition 1. *Global non-interference means that for any two executions with corresponding objects and futures, such that the history of messages consumed or produced by an object in one execution state is indistinguishable from that of the corresponding object in a state of the other execution, and such that the next communication event of the first object is a low output, then the next low communication output event of the other object will be indistinguishable.*

Definition 2. *Local non-interference means that for two executions with corresponding objects o and o', and for execution states where o and o' are non-idle and where the execution states of o and o' are indistinguishable, the next execution states of these objects will also be indistinguishable when both have executed*

the next statement, and in case the statement gives an output, both make indistinguishable output (or neither makes no low output).

Note that our security approach includes termination aspects. We next prove that each object is *locally deterministic*, in the sense that the next state of a statement, other than idle and get, is deterministic, i.e., depending only on the prestate. The only source of non-determinism is get and the independent speed of the objects, which means that the ordering in the messages queues is in general non-deterministic. Thus only idle states and get cause local non-determinism.

Lemma 1. *In our security model, each object is locally deterministic.*

Proof. According to our operational semantics, for each statement (other than idle and get) there is only one rule to apply, and for an `if` statement, the choice of the rule is given deterministically by testing the security level and value of the guard. There is no interleaving of processes inside an object as well. □

Definition 3. *Low-to-low determinism means that any low part of a state or output resulting from a statement, other than get, is determined by the low part of the prestate and the statement, when ignoring states where pcs is high.*

Lemma 2. *In our security model, each object is low-to-low deterministic.*

Proof. This can be proved by case analysis on the statements. For an `if` with a high test, the taken branch does not result in low state changes nor low outputs. In particular, any invocation message made has label H, and the execution of that method invocation by the same or another object, will start in a high context (see the START-FUT and START rules), and so will a new object created from the branch. This ensures that there is no implicit leakage from a high branch. However, the choice of branch could depend on high information, and lead to distinguishable states, but this is compensated by $endif(s'')$, which raises the level of variables updated in the untaken branch s''. For an `if` with low test, the choice of branch is given by the low part of the prestate and the test. For an assignment, the level of the left-hand-side becomes low if the level of the right-hand-side is low and *pcs* is low. Otherwise, the left-hand-side' level becomes high after the assignment. The cases for the other statements are straightforward. □

Theorem 1. *Our security model guarantees local and global non-interference, and an attacker (i.e., a low object) will only receive low information.*

Proof. Local non-interference can be proved by induction of the number of execution steps considering two executions of an object. The low part of each state and the low outputs must be the same by the two previous lemmas, using the fact that future values of corresponding futures will be indistinguishable, since these are given by earlier outputs, which are indistinguishable by the induction hypothesis. Global non-interference can be proved by induction on the number of steps considering two executions. It follows by local non-interference for all objects. Since an attacker is a low object, the wrappers will prevent it from receiving high inputs. □

This theorem implies that an attacker will not be able to obtain high information explicitly or implicitly, nor obvserve difference of termination aspects.

5 Related work

Starting with the work of Denning and Denning [9], a number of static techniques for lattice-based security information flow analysis have been suggested.

In [20], a secure type system has been suggested for Creol without futures to enforce noninterference in object interactions. Typing rules check that the security levels of variables respect the declared security levels in the interfaces. In [20], since the run-time security levels of objects, indicating the access rights, might not be available at static time, an if-test construct is added to check the security level of an object before sending data. Our approach is a dynamic technique, which is more permissive and precise and supports futures confidentiality. In [22], Pettai and Laud present a type system for ABS to ensure non-interference by means of over-approximation. E.g., a future's security level is the upper bound of the tasks' levels that the future refers to, while our run-time system does not use over-approximation (assuming the labels are exact). This work also deals with other concurrency features of ABS such as cogs and synchronization between tasks, where security issues are prevented by using the operational semantics and the type system. The cog feature of ABS is not relevant to our paper.

In [2], a dynamic information-flow control approach is performed for the ASP language. Security levels are assigned to activities and communicated data. The security levels do not change when they are assigned. Dynamic checks are performed at activity creations, requests, and replies. Since future references are not confidential, they are passed between activities without dynamic checking, but getting a future value is checked by a reply transmission rule. In [2], the security model guarantees data confidentiality for multi-level security (MLS) systems. Our approach adds flow-sensitivity, which allows security levels of variables to change during execution of an object. It makes our approach more permissive and a wrapper deals with run-time security levels. In addition to enforcing the non-interference property in object interactions, our approach guarantees that an object will be given access only to the information that it is allowed to handle.

In [18], Nair et al. implement and design a run-time system, named Trishul, to track the flow of information within the Java virtual machine (JVM). This paper focuses on implicit and explicit flows through the Java control flows and the architecture and does not enforce non-interference. Due to the Trishul's modular nature, our security wrappers can be deployed to prevent illegal flows.

Russo and Sabelfeld [23] prove that a sound flow-sensitive dynamic information-flow enforcement is more permissive than static analysis. In [16], the notion of wrappers is used to control the behavior of JavaScript programs and enforce security policies to protect web pages from malicious codes. A policy specifies under which conditions a page performs a specific action, and a wrapper grants, rejects, or modifies these actions. Moreover, the notion of wrappers has been developed for the safety of objects [21], where the programmer needs to specify which objects should have a wrapper and to program what each wrapper should do based on any input/output. In contrast, we apply wrappers to security analysis, letting the runtime system automatically decide which components should be wrapped, and also what the wrappers should do to prevent illegal flows.

6 Conclusion

We have proposed a framework for enforcing secure information-flow and non-interference in active object languages based on the notion of security wrappers. We have considered a high-level core language supporting asynchronous calls and futures. In our model, due to encapsulation, there is no need for information-flow restrictions inside an object. Wrappers perform security checks for object interactions (with methods and futures) at run-time. Furthermore, wrappers control the access to futures with high values. Security rules of wrappers are defined based on security levels of communicated messages. Inside an object, the security levels of variables might change at run-time due to flow-sensitivity. Wrappers on unsafe objects and future components protect exchange of confidential values to low objects. Wrappers on objects protect outgoing method calls and prevent leakage of information through outgoing parameters. The wrappers are created automatically by the run-time system without the involved parties being aware of it. Their behavior is also defined by the runtime system. We define non-interference for our language and outline a proof of it. By combining results from static analysis, we can improve run-time efficiency by avoiding wrappers when they are superfluous according to the over-approximation of levels given by the static analysis.

Acknowledgements. We thank Christian Johansen for useful interactions. The Norwegian Research Council has funded us by project *IoTSec* (no. 248113/O70).

References

1. Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, Massachusetts Inst. of Tech, Cambridge Artificial Int. Lab., 1985.
2. Isabelle Attali, Denis Caromel, Ludovic Henrio, and Felipe Luna Del Aguila. Secured information flow for asynchronous sequential processes. *Electronic Notes in Theoretical Computer Science*, 180(1):17–34, 2007.
3. Henry C Baker Jr and Carl Hewitt. The incremental garbage collection of processes. *ACM Sigplan Notices*, 12(8):55–59, 1977.
4. Musard Balliu, Daniel Schoepe, and Andrei Sabelfeld. We are family: Relating information-flow trackers. In *European Symposium on Research in Computer Security*, pages 124–145. Springer, 2017.
5. Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Computing Surveys*, 50(5):76, 2017.
6. Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, S Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. Parallel objects for multicores: A glimpse at the parallel language Encore. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, volume 9104 of Lecture Notes in Computer Science, pages 1–56. Springer, 2015.
7. Denis Caromel, Christian Delbé, Alexandre Di Costanzo, and Mario Leyton. Proactive: an integrated platform for programming and running applications on Grids and P2P systems. *Computational Methods in Science & Technology*, 12.1:16, 2006.

8. Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects: Asynchrony-Mobility-Groups-Components*. Springer, 2005.
9. Dorothy E Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
10. Joseph A Goguen and José Meseguer. Security policies and security models. In *Security and Privacy, 1982 IEEE Symposium on*, pages 11–11. IEEE, 1982.
11. Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410:202–220, 2009.
12. Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. *Software Safety and Security*, 33:319–347, 2012.
13. Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*, volume 6957 of Lecture Notes in Computer Science, pages 142–164. Springer, 2011.
14. Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software & Systems Modeling*, 6(1):39–58, 2007.
15. Farzane Karami, Olaf Owe, and Toktam Ramezanifarkhani. An evaluation of interaction paradigms for active objects. *Journal of Logical and Algebraic Methods in Programming*, 103:154 – 183, 2019.
16. Jonas Magazinius, Phu H Phung, and David Sands. Safe wrappers and sane policies for self protecting Javascript. In *Nordic Conference on Secure IT Systems*, pages 239–255. Springer, 2010.
17. José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical computer science*, 96(1):73–155, 1992.
18. Srijith K Nair, Patrick ND Simpson, Bruno Crispo, and Andrew S Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science*, 197(1):3–16, 2008.
19. Olaf Owe and Ole-Johan Dahl. Generator induction in order sorted algebras. *Formal Aspects Comput.*, 3(1):2–20, 1991.
20. Olaf Owe and Toktam Ramezanifarkhani. Confidentiality of interactions in concurrent object-oriented systems. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, volume 10436 of Lecture Notes in Computer Science, pages 19–34. Springer, 2017.
21. Olaf Owe and Gerardo Schneider. Wrap your objects safely. *Electronic Notes in Theoretical Computer Science*, 253(1):127–143, 2009.
22. Martin Pettai and Peeter Laud. Securing the future — an information flow analysis of a distributed OO language. In *SOFSEM 2012: Theory and Practice of Computer Science*, pages 576–587. Springer, 2012.
23. Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 186–199. IEEE, 2010.
24. Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.
25. Marjan Sirjani, Ali Movaghar, and Mohammad Reza Mousavi. Compositional verification of an object-based model for reactive systems. In *Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'01), Oxford, UK*, pages 114–118. Citeseer, 2001.
26. Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S De Boer. Modeling and verification of reactive systems using Rebeca. *Fundamenta Informaticae*, 63(4):385–410, 2004.
27. Derek Wyatt. *Akka concurrency*. Artima Incorporation, 2013.